DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN
ESCUELA DE INGENIERÍA
PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

IIC2343 - Arquitectura de Computadores (I/2026)

Ayudantía 7: Subrutinas

Ayudantes: Alberto Maturana (alberto.maturana@uc.cl), Mario Rojas (mario.denzel@estudiante.uc.cl, Nicolás Romo Aubele (nroma@uc.cl)

Pregunta 1: Preguntas conceptuales

a) En el computador básico del curso, la memoria de *stack* se utiliza para soportar la ejecución de subrutinas. Describa cómo está organizada esta memoria y el rol del registro `SP` (*Stack Pointer*). Luego, explique por qué es fundamental inicializar correctamente el `SP` y analice las consecuencias de una inicialización incorrecta. Finalmente, discuta cómo estos problemas se ven acentuados en presencia de llamadas recursivas y explique en qué consiste un *stack overflow* en este contexto.

b) Considera el siguiente código en *assembly* del computador básico. Explique el propósito, y cómo funciona, la secuencia de instrucciones `PUSH B`, `CALL func` y `POP B`; y cuál es el efecto de que no haya una instrucción `PUSH A` justo antes, ni una instrucción `POP A` justo después de la llamada a la función `func`.

```
DATA:
    r 3
CODE :
    MOV  A, (r)
    MOV  B, 2
    PUSH  B
    CALL  func
    POP  B
    JMP  end
    func:
        MOV A, B
        ADD A, 3
        MOV B, A
        MOV (r), B
        RET
    end:
```

Solución: a) En el computador básico, la memoria de *stack* corresponde a una región de la memoria de datos ubicada en el segmento inferior, utilizada para almacenar información temporal como direcciones de retorno y valores de registros. El acceso se realiza mediante el registro `SP` (*Stack Pointer*), que se inicializa en la última dirección de memoria (por ejemplo, 255 en una memoria de 8 *bits*). A medida que se ejecutan instrucciones como `PUSH` o `CALL`, el *stack* crece hacia direcciones menores, decrementando el valor de `SP`.

Inicializar correctamente el `SP` es fundamental para asegurar que el *stack* tenga suficiente espacio para crecer sin interferir con las variables del programa, manteniendo separadas ambas regiones de memoria.

Si el `SP` se inicializa incorrectamente, por ejemplo en la dirección 0, o en la 64, pueden producirse errores graves. Ya que el *stack* podría sobrescribir variables del programa, generando corrupción de datos, y las instrucciones `CALL` y `RET` pueden utilizar direcciones de retorno incorrectas, afectando el flujo de ejecución.

Este problema se vuelve aún más crítico en presencia de recursión, donde múltiples llamadas a una misma subrutina se acumulan en el *stack*. Cada llamada recursiva requiere almacenar su propia dirección de retorno y posibles datos adicionales, por lo que el uso del *stack* crece rápidamente. Si el espacio disponible es insuficiente —ya sea por una mala inicialización del `SP` o por una gran profundidad de recursión— puede ocurrir *stack overflow*, donde el *stack* colisiona con otras regiones de memoria. Esto provoca comportamiento incorrecto, pérdida de información o incluso la detención del programa.

b) El propósito es ejecutar la subrutina `func`, pero de modo que el valor del registro `B` al terminar la ejecución de `func` y volver al programa principal `(CODE)` sea el mismo que `B` tenía justo antes de hacer la llamada; es decir, las actualizaciones del registro `B` al interior de `func` no se ven reflejadas en `CODE`.

Esto ocurre porque el valor de `B` justo antes de la llamada es guardado en el *stack*, mediante `PUSH B`, y es restaurado justo después de la llamada mediante `POP B`. Por esto mismo, la ausencia de las instrucciones `PUSH A` y `POP A` tiene el efecto contrario en el caso del registro `A`: los cambios que el registro `A` sufre dentro de *func* se ven reflejados en `CODE` cuando `func` termina su ejecución.

Las instrucciones `PUSH` y `POP` acualizan el valor del *stack pointer* — justo después y justo antes, respectivamente — de tener acceso a la dirección de memoria apuntada por el *stack pointer*.

Pregunta 2: Análisis de código

2.1.a) (I2 2025-2) Para el siguiente fragmento de código indique los valores de los registros `A` y `B` al finalizar la ejecución del código.

2.1.b) Para el mismo fragmento de código, indique los valores de los registros `A` y `B` al finalizar la ejecución del código, pero asumiendo que la línea con la instrucción `RET` no es parte del programa.

```
DATA:
    var1 12
    var2 4
CODE:
    MOV A,5
    MOV B,3
    RET
    case_1:
        ADD A,B // Dir. memoria 0x04
        PUSH A  // Dir. memoria 0x05
        MOV A,B // Dir. memoria 0x06
        SHL A,A // Dir. memoria 0x07
        MOV B,A // Dir. memoria 0x08
        POP A   // Dir. memoria 0x09-0x0A
        JMP end // Dir. memoria 0x0B
    case_2:
        SUB A,B // Dir. memoria 0x0C
        PUSH A  // Dir. memoria 0x0D
        MOV A,B // Dir. memoria 0x0E
        SHR A,A // Dir. memoria 0x0F
        MOV B,A // Dir. memoria 0x10
        POP A   // Dir. memoria 0x11-0x12
    end:
```

2.2) Se describirán nuevas instrucciones, considere que ya están implementadas en un computador básico con subrutinas. Considere que cada instrucción se ejecuta en un ciclo.

- `CALL B,Dir`: Almacena en el registro B la dirección de retorno PC+1 y salta a la dirección de memoria equivalente al literal Dir.
- `RET B`: Salta a la dirección de retorno almacenada en el registro B.

A continuación, se adjunta el siguiente fragmento de código escrito en el *Assembly* del computador básico, con la diferencia de que incorpora las nuevas instrucciones implementadas. Para cada fragmento, indique los valores de los registros A y B al finalizar la ejecución del código.

```
DATA:
    n 2
    res 0
CODE:
    MOV A,(n)           // Dir. mem. 0x00
    CALL B,fun          // Dir. mem. 0x01
    JMP end             // Dir. mem. 0x02
        fun:
            CMP A,0     // Dir. mem. 0x03
            JLE fun_end // Dir. mem. 0x04
            PUSH B      // Dir. mem. 0x05
            MOV B,A     // Dir. mem. 0x06
            ADD A,(res) // Dir. mem. 0x07
            MOV (res),A // Dir. mem. 0x08
            MOV A,B     // Dir. mem. 0x09
            SUB A,1     // Dir. mem. 0x0A
            CALL B,fun  // Dir. mem. 0x0B
            POP B       // Dir. mem. 0x0C-0x0D
        fun_end:
            RET B       // Dir. mem. 0x0E

    end:
        MOV A,(res)     // Dir. mem. 0x0F
```

Solución:

2.1.a) El fragmento de código accede, por overflow del registro `SP`, a la dirección de memoria 0 (equivalente a `var1`) para cargar su valor en el registro PC, por lo que la ejecución de `RET` genera el salto al label `case_2`. A continuación, se muestra el resultado de la ejecución del fragmento:

```
DATA:
    var1 12 // MEM[0] =12
    var2 4  // MEM[1] =4
CODE:
    MOV A,5 // A= 5
    MOV B,3 // B= 3
    RET     // SP++ =>255 +1= 0,PC= MEM[0]= 12
    case_1:
        ADD A,B // NO SE EJECUTA
        PUSH A  // NO SE EJECUTA
        MOV A,B // NO SE EJECUTA
        SHL A,A // NO SE EJECUTA
        MOV B,A // NO SE EJECUTA
        POP A   // NO SE EJECUTA
        JMP end // NO SE EJECUTA
    case_2:
        SUB A,B // A= 2
        PUSH A  // MEM[SP]= A=>MEM[0]=2, SP--=>0-1
     =255
        MOV A,B // A= 3
        SHR A,A // A= 1
        MOV B,A // B= 1
        POP A   // SP++ =>255 +1= 0,A=MEM[0]= 2
    end:        // A=2, B=1
```

2.1.b) A diferencia del código anterior, dado que la instrucción `RET` ya no forma parte del programa, se entra directamente al `case_1`, y dado que existe un `JMP` antes de `case_2`, entonces esta vez no se ejecuta esta parte del código saltando directamente al final de este. A continuación, se muestra el resultado de la ejecución del fragmento:

```
DATA:
    var1 12 // MEM[0] =12
    var2 4  // MEM[1] =4
CODE:
    MOV A,5 // A= 5
    MOV B,3 // B= 3
    case_1:
        ADD A,B // A = 8
        PUSH A  // MEM[SP]= A=>MEM[0]=8, SP--=>0-1
     =255
        MOV A,B // A = 3
        SHL A,A // A = 6
        MOV B,A // B = 6
        POP A   // SP++ => 255 + 1 = 0,A=MEM[0] = 8
        JMP end // Va al al final
    case_2:
        SUB A,B // NO SE EJECUTA
        PUSH A  // NO SE EJECUTA
        MOV A,B // NO SE EJECUTA
        SHR A,A // NO SE EJECUTA
        MOV B,A // NO SE EJECUTA
        POP A   // NO SE EJECUTA
    end:        // A=8, B=6
```

2.2) El fragmento de código computa de forma recursiva la suma de la variable `n` con todos sus antecesores hasta llegar a cero, almacenando el resultado final en `res`. A continuación, se muestra el resultado de la ejecución del fragmento:

```
DATA:
    n 2                 // MEM[0]=2
    res 0               // MEM[1]=0
CODE:
    MOV A,(n)           // A = n = 2
    CALL B,fun          // B = 2, fun(n)
    JMP end
    fun:
        CMP A,0
        JLE fun_end     // Termina cuando A = 0
        PUSH B          // Respalda B en stack (retorno)
        MOV B,A         // B = A = n
        ADD A,(res)     // A += res
        MOV (res),A     // res = res + n
        MOV A,B         // A = n
        SUB A,1         // n--
        CALL B,fun      // fun(n-1)
        POP B           // Recupera B (retorno)
    fun_end:
        RET B           // Retorna de la forma esperada
    end:
    // A = res = fun(2) = 2+fun(1) = 2+1+fun(0) = 2+1+0 = 3
        MOV A,(res)     // B = Primer retorno computado = 2
```

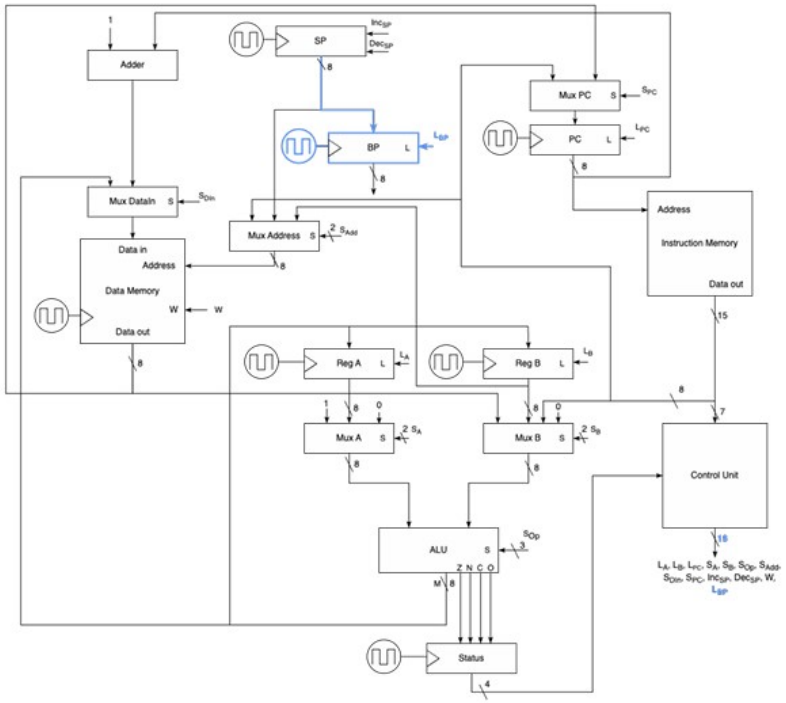
Pregunta 3: Modificaciones al computador básico

En varias arquitecturas se hace uso del registro *base pointer* (`BP`). Este funciona como punto de referencia para obtener los parámetros y dirección de retorno de cada llamada de una subrutina, lo que facilita el manejo de llamados anidados. Deberá modificar la microarquitectura del computador básico para implementar el registro `BP` de 8 *bits* y su funcionamiento. Se le listarán las instrucciones que deben ser implementadas. Para cada una de ellas, debe incluir la combinación completa de señales que las ejecutan. En las señales de carga/escritura/incremento/decremento, debe indicar si se activan (1) o no (0); en las señales de selección, debe indicar el nombre de la entrada escogida ("-" si no afecta). Puede realizar todas las modificaciones en un solo diagrama.

- `MOV BP, SP`. Almacena en el registro `BP` el valor del registro `SP`.
- `PUSH BP`. Almacena en `Mem[SP]` el valor de `BP` y decrementa `SP` en una unidad.
- `POP BP`. Incrementa `SP` en una unidad y almacena en `BP` el valor `Mem[SP]`.
- `MOV Reg,(BP + Lit)`. Almacena en `Reg (A o B)` el valor `Mem[BP+Lit]`.

Solución:

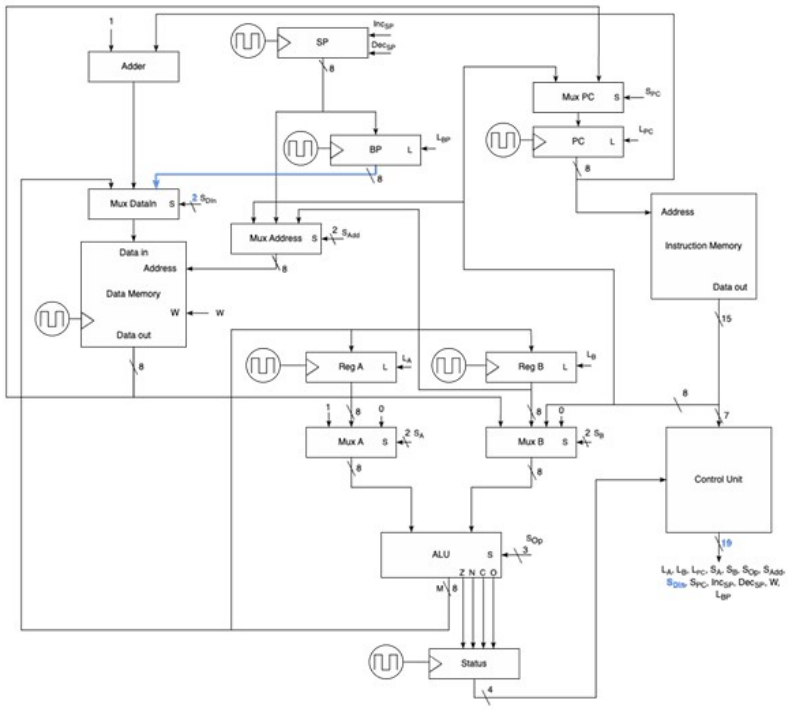
a) Para esta modificación, agregamos el registro `BP` a nuestra arquitectura con una nueva señal de carga L_{BP} y con valor de entrada igual a la salida de `SP`. De momento, no conectamos la salida de `BP` con ninguna compuerta (se asume que va a tierra).



La combinación de instrucciones para ejecutar la instrucción es:

Instrucción	L_A	L_B	L_{PC}	W	Inc_{SP}	Dec_{SP}	S_A	S_B	S_{OP}	S_{Add}	S_{DIn}	S_{PC}	L_{BP}
MOV BP,SP	0	0	0	0	0	0	-	-	-	-	-	-	1

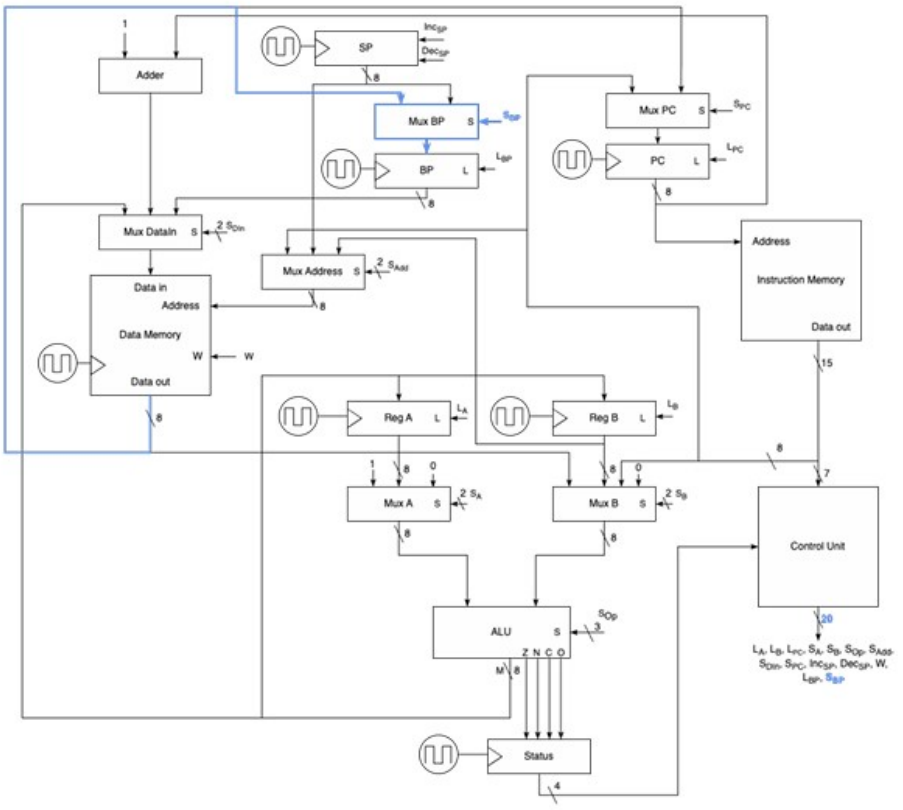
b) Para esta modificación, conectamos la salida del registro BP con una de las entradas del componente Mux DataIn. Para ello, es necesario incrementar la cantidad de bits de la señal S_{DIn}) a dos.



La combinación de instrucciones para ejecutar la instrucción es:

Instrucción	L_A	L_B	L_{PC}	W	Inc_{SP}	Dec_{SP}	S_A	S_B	S_{OP}	S_{Add}	S_{DIn}	S_{PC}	L_{BP}
PUSH BP	0	0	0	1	0	1	-	-	-	SP	BP	-	0

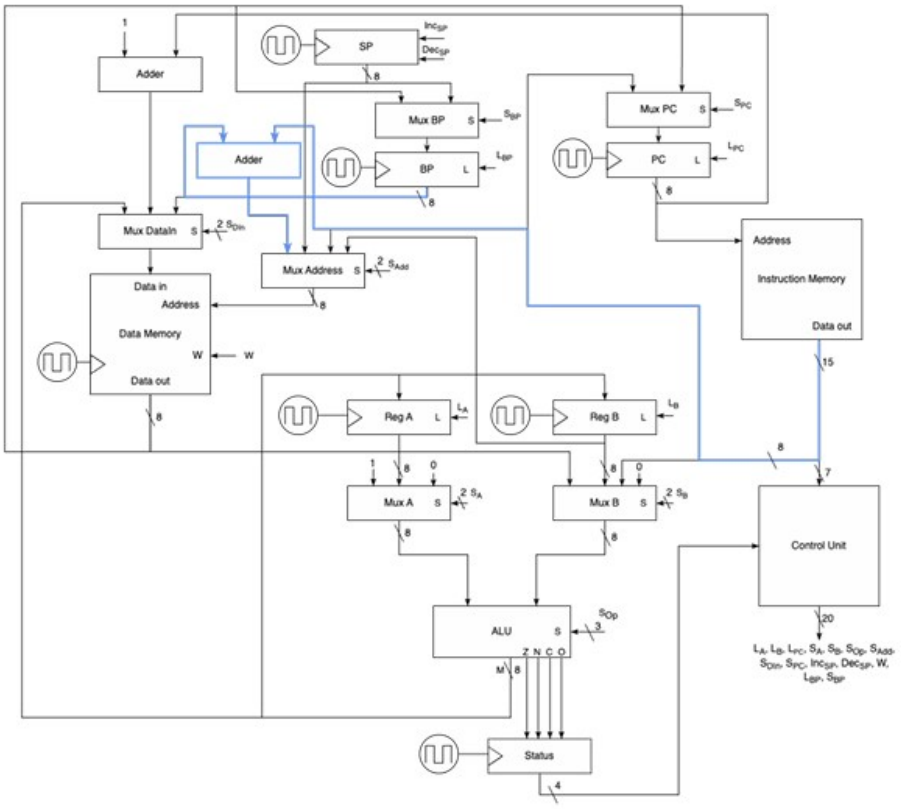
c) Para esta modificación, conectamos la salida de la memoria de datos con la entrada de BP. Dado que ya existe una entrada para SP, se agrega un nuevo componente Mux BP con señal de selección S_{BP} para escoger el valor de carga entre SP y DOUT.



La combinación de instrucciones para ejecutar la instrucción es:

Instrucción	L_A	L_B	L_{PC}	W	Inc_{SP}	Dec_{SP}	S_A	S_B	S_{OP}	S_{Add}	S_{DIn}	S_{PC}	L_{BP}	S_{BP}
POP BP	0	0	0	0	1	0	-	-	-	-	-	-	0	-
	0	0	0	0	0	0	-	-	-	SP	-	-	1	DOUT

d) Para esta modificación, agregamos un `Adder` (sumador) que recibirá como entradas el valor del registro `BP` y el literal `LIT` de la memoria de instrucciones, para luego conectar su salida al componente `Mux Address`.



La combinación de instrucciones para ejecutar la instrucción es:

Instrucción	L_A	L_B	L_{PC}	W	Inc_{SP}	Dec_{SP}	S_A	S_B	S_{OP}	S_{Add}	S_{DIn}	S_{PC}	L_{BP}	S_{BP}
MOV A,(BP + Lit)	1	0	0	0	0	0	0	DOUT	ADD	BP+LIT	-	-	0	-
MOV B,(BP + Lit)	0	1	0	0	0	0	0	DOUT	ADD	BP+LIT	-	-	0	-

Instrucción	Operandos	*Opcode*	Condición	L_{PC}	L_A	L_B	S_A0,1	S_B0,1	S_{Op}0,1,2	S_{Add}0,1	S_{Din}	S_{PC}	W	Inc_{SP}	Dec_{SP}
MOV	A,B	0000000		0	1	0	ZERO	B	ADD	-	-	-	0	0	0
	B,A	0000001		0	0	1	A	ZERO	ADD	-	-	-	0	0	0
	A,Lit	0000010		0	1	0	ZERO	LIT	ADD	-	-	-	0	0	0
	B,Lit	0000011		0	0	1	ZERO	LIT	ADD	-	-	-	0	0	0
	A,(Dir)	0000100		0	1	0	ZERO	DOUT	ADD	LIT	-	-	0	0	0
	B,(Dir)	0000101		0	0	1	ZERO	DOUT	ADD	LIT	-	-	0	0	0
	(Dir),A	0000110		0	0	0	A	ZERO	ADD	LIT	ALU	-	1	0	0
	(Dir),B	0000111		0	0	0	ZERO	B	ADD	LIT	ALU	-	1	0	0
	A,(B)	0001000		0	1	0	ZERO	DOUT	ADD	B	-	-	0	0	0
	B,(B)	0001001		0	0	1	ZERO	DOUT	ADD	B	-	-	0	0	0
	(B),A	0001010		0	1	0	A	ZERO	ADD	B	ALU	-	1	0	0
ADD	A,B	0001011		0	1	0	A	B	ADD	-	-	-	0	0	0
	B,A	0001100		0	0	1	A	B	ADD	-	-	-	0	0	0
	A,Lit	0001101		0	1	0	A	LIT	ADD	-	-	-	0	0	0
	A,(Dir)	0001110		0	1	0	A	DOUT	ADD	LIT	-	-	0	0	0
	A,(B)	0001111		0	1	0	A	DOUT	ADD	B	-	-	0	0	0
	(Dir)	0010000		0	0	0	A	B	ADD	LIT	ALU	-	1	0	0
SUB	A,B	0010001		0	1	0	A	B	SUB	-	-	-	0	0	0
	B,A	0010010		0	0	1	A	B	SUB	-	-	-	0	0	0
	A,Lit	0010011		0	1	0	A	LIT	SUB	-	-	-	0	0	0
	A,(Dir)	0010100		0	1	0	A	DOUT	SUB	LIT	-	-	0	0	0
	A,(B)	0010101		0	1	0	A	DOUT	SUB	B	-	-	0	0	0
	(Dir)	0010110		0	0	0	A	B	SUB	LIT	ALU	-	1	0	0
AND	A,B	0010111		0	1	0	A	B	AND	-	-	-	0	0	0
	B,A	0011000		0	0	1	A	B	AND	-	-	-	0	0	0
	A,Lit	0011001		0	1	0	A	LIT	AND	-	-	-	0	0	0
	A,(Dir)	0011010		0	1	0	A	DOUT	AND	LIT	-	-	0	0	0
	A,(B)	0011011		0	1	0	A	DOUT	AND	B	-	-	0	0	0
	(Dir)	0011100		0	0	0	A	B	AND	LIT	ALU	-	1	0	0
OR	A,B	0011101		0	1	0	A	B	OR	-	-	-	0	0	0
	B,A	0011110		0	0	1	A	B	OR	-	-	-	0	0	0
	A,Lit	0011111		0	1	0	A	LIT	OR	-	-	-	0	0	0
	A,(Dir)	0100000		0	1	0	A	DOUT	OR	LIT	-	-	0	0	0
	A,(B)	0100001		0	1	0	A	DOUT	OR	B	-	-	0	0	0
	(Dir)	0100010		0	0	0	A	B	OR	LIT	ALU	-	1	0	0
NOT	A,A	0100011		0	1	0	A	-	NOT	-	-	-	0	0	0
	B,A	0100100		0	0	1	A	-	NOT	-	-	-	0	0	0
	(Dir)	0100101		0	0	0	A	B	NOT	LIT	ALU	-	1	0	0
XOR	A,B	0100110		0	1	0	A	B	XOR	-	-	-	0	0	0
	B,A	0100111		0	0	1	A	B	XOR	-	-	-	0	0	0
	A,Lit	0101000		0	1	0	A	LIT	XOR	-	-	-	0	0	0
	A,(Dir)	0101001		0	1	0	A	DOUT	XOR	LIT	-	-	0	0	0
	A,(B)	0101010		0	1	0	A	DOUT	XOR	B	-	-	0	0	0
	(Dir)	0101011		0	0	0	A	B	XOR	LIT	ALU	-	1	0	0
SHL	A,A	0101100		0	1	0	A	-	SHL	-	-	-	0	0	0
	B,A	0101101		0	0	1	A	-	SHL	-	-	-	0	0	0
	(Dir)	0101110		0	0	0	A	B	SHL	LIT	ALU	-	1	0	0
SHR	A,A	0101111		0	1	0	A	-	SHR	-	-	-	0	0	0
	B,A	0110000		0	0	1	A	-	SHR	-	-	-	0	0	0
	(Dir)	0110001		0	0	0	A	B	SHR	LIT	ALU	-	1	0	0
INC	B	0110010		0	0	1	ONE	B	ADD	-	-	-	0	0	0
CMP	A,B	0110101		0	0	0	A	B	SUB	-	-	-	0	0	0
	A,Lit	0110110		0	0	0	A	LIT	SUB	-	-	-	0	0	0
JMP	Dir	0111001		1	0	0	-	-	-	-	-	LIT	0	0	0
JEQ	Dir	0111010	Z=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JNE	Dir	0111011	Z=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JGT	Dir	0111100	N=0 y Z=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JLT	Dir	0111101	N=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JGE	Dir	0111110	N=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JLE	Dir	0111111	N=1 o Z=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JCR	Dir	1000000	C=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JOV	Dir	1000001	V=1	1	0	0	-	-	-	-	-	LIT	0	0	0
CALL	Dir	1000010		1	0	0	-	-	-	SP	PC	LIT	1	0	1
RET		1000011		0	0	0	-	-	-	-	-	-	0	1	0
		1000100		1	0	0	-	-	-	SP	-	DOUT	0	0	0
PUSH	A	1000101		0	0	0	A	ZERO	ADD	SP	ALU	-	1	0	1
	B	1000110		0	0	0	ZERO	B	ADD	SP	ALU	-	1	0	1
POP	A	1000011		0	0	0	-	-	-	-	-	-	0	1	0
		1000111		0	1	0	ZERO	DOUT	ADD	SP	-	-	0	0	0
POP	B	1000011		0	0	0	-	-	-	-	-	-	0	1	0
		1001000		0	0	1	ZERO	DOUT	ADD	SP	-	-	0	0	0
NOP		1001001		0	0	0	-	-	-	-	-	-	0	0	0

Tabla 1: ISA del computador básico.

Feedback ayudantía

Escanee el QR para entregar feedback sobre la ayudantía.

